

# Computer Science Tripos

## Part II Project Proposal Coversheet

Please fill in Part 1 of this form and attach it to the front of your Project Proposal.

### Part 1

Name:  CRSID:

College:  Overseers: (Initials)

Title of Project:

Date of submission:  Will Human Participants be used?

Project Originator:  Signature: -----

Project Supervisor:  Signature: -----

Directors of Studies:  Signature: -----

Special Resource Sponsor:  Signature: -----

Special Resource Sponsor:  Signature: -----

***Above signatures to be obtained by the Student***

### Part 2

Overseer Signature 1: -----

Overseer Signature 2: -----

***Overseers signatures to be obtained by Student Administration.***

Overseers Notes:

### Part 3

SA Date Received:  SA Signature Approved:

## A Prolog Compiler and Abstract Machine

Phoebe Nichols, pmn29  
Churchill College  
October 2018

<b>Project Originator:</b>	Phoebe Nichols	<b>Project Supervisor:</b>	Prof. Alan Mycroft
<b>Director of Studies:</b>	Dr John Fawcett	<b>Overseers:</b>	Prof. Andrew Pitts Dr Rafal Mantiuk

### Introduction and Description of the Work

Prolog is a declarative programming language. This means that a Prolog programmer declares rules that describe the result of a program, but does not specify how the program should achieve this result. This is in contrast to imperative programming languages such as Java, where the programmer specifies how a program's state should be changed in order to achieve the result of the program. The use of a declarative language presents the interesting problem of how to evaluate a query without being told how the answer to the query should be computed. In the case of Prolog, this problem is solved by performing a graph search over the Horn clauses given in a Prolog program. This graph search attempts to find an assignment of values to variables that satisfies the query, or rejects the query if no such assignment exists. Typically, a left-to-right depth-first search is performed.

The aim of this project is to implement a compiler and abstract machine for Prolog. The target for the Prolog compiler will be specialised byte-code. Designing this byte-code will form part of the project. The abstract machine implemented will interpret this byte-code to run the original Prolog program. An existing specification of an abstract machine to execute Prolog, the Warren Abstract Machine (WAM), will be used as inspiration—but I will not implement the full WAM.

The project will be restricted to a basic subset of Prolog so that it remains feasible. Some extensions that I am considering (for example type checking) will require an extension to Prolog syntax in order to provide information about program properties. The syntax I use may therefore not be a true subset of standard Prolog.

### Starting Point

#### Prolog as a programming language

I will need a good understanding of how to program in Prolog in order to implement the Prolog compiler. I will also need to write test programs for the subset of Prolog that I have chosen to compile. These tests will help me to evaluate the correctness and performance of my compiler and abstract machine implementation.

I am studying the 50% course, which does not cover Prolog until Lent term this year. I have, therefore, studied the Prolog course over the summer.

## Prolog compilation and execution

I have no prior experience with implementing Prolog interpreters or compilers. The IB 75%/II 50% Prolog course that I studied over the summer introduced me to the basics of Prolog execution. I have also read around this area over the summer: some of the papers I read are cited in this document.

## OCaml

I am intending to implement all of my project in OCaml initially, although I may add some SWI-Prolog for extension features. I have chosen OCaml because I have experience implementing lexers and parsers in ML from the compilers course, and found ML convenient for this sort of thing. OCaml will have same advantages as ML, but is more widely used and therefore has more tools and libraries available.

I have very little prior experience with OCaml (only from the IB compilers course), so I will need to dedicate time to studying it. This time is provisioned in my project plan.

## Structure of the Project

The project is made up of four main components:

1. Lexer
2. Parser
3. Translator
4. Abstract machine

I will implement the components in this order. This is because each component can be used to generate test cases for the next component. As an extension, I may add a semantic analysis phase acting on the parse tree. This would verify program properties and try to provide information so that the translator can generate more efficient code.

## Testing

Unit tests will be written for each individual component. Components written in OCaml will likely have unit tests written with the testing framework OUnit [1]. Any components written in SWI-Prolog will be tested using native support for unit tests as described in the SWI-Prolog documentation [2]. I will write unit tests for each component along with the implementation of the component itself.

## Evaluation

I will be able to quantitatively evaluate the project in terms of:

- The run-time performance of my abstract machine compared to standard Prolog implementations such as SWI-Prolog.
- The run-time performance of my abstract machine compared to my interpreter.
- The change in code size, number of choice points, or performance, for any optimisations that I implement.

Here, ‘performance’ means a comparison of space and time usage for sample programs. The Spacetime functionality of the OCaml compiler can be used for profiling the memory behaviour of an OCaml program [3], and SWI-Prolog includes predicates for execution profiling [4].

## Success Criterion

My success criterion is to implement the following components for a basic subset of Prolog:

- Lexer
- Parser
- Translator from parse tree to byte-code
- Abstract machine to execute this byte-code

These components should enable me to correctly execute example programs written in the grammar that I use. The components will not implement the full ISO Prolog standard, and the grammar used may include features such as type annotations not found in standard Prolog. The lexer and parser might be implemented manually, or using `ocamllex` and `ocamlyacc`.

## Possible extensions

The main extensions for this project involve adding optimisations to the compiler. Some potential optimisations are:

- **Last call optimisation**  
Tail recursion can be converted to iteration to save adding a new stack frame for each tail recursive call.
- **Determinacy analysis**  
A clause is said to be determinate if it can only return one possible solution [5]. This information can be used to avoid backtracking through determinate clauses.
- **Mode analysis**  
The argument to a predicate can be assigned a mode to say that the argument is always used as input, or as output [6]. Mode information can be used to invoke special purpose unification routines that test for fewer cases and therefore can be faster.
- **Type checking**  
It is possible to define a polymorphic type system for Prolog [7]. Type information can be used to select faster, special-purpose unification routines. A type checker also has the significant advantage of potentially spotting program bugs and saving developer time.

There are many more possible optimisations I could apply, and I might also select one of these for an extension. As I implement the initial system I will learn more about the optimisations I could make, and may also find my implementation to be more suited to certain optimisations.

## Timetable and Milestones

**22<sup>nd</sup> Oct – 4<sup>th</sup> Nov**

**19<sup>th</sup> Oct:** *Proposal submitted*

Project set-up:

- Investigate and set up an IDE for OCaml.
- Set up a backup system for the project.
- Choose a software engineering methodology to follow.
- Learn to use the testing frameworks needed for the project.

Write small programs in SWI-Prolog, to re-enforce understanding of Prolog and for potential use as benchmarks. Study OCaml using the Real World OCaml textbook [8], and write small programs in OCaml to gain familiarity with the language.

Milestones:

- Document produced to describe the development environment, backup system, software engineering methodology, and testing frameworks used.
- Example programs written in OCaml.
- Example programs written in Prolog.

### **5<sup>th</sup> Nov – 18<sup>th</sup> Nov**

Implement a lexer and parser for the chosen subset of Prolog. I am intending to use `ocamllex` and `ocamlyak` (described in [9]) for this.

Milestones:

- Code for lexer written and tested.
- Code for parser written and tested.
- Document produced to describe the implementation of these components.

### **19<sup>th</sup> Nov – 2<sup>nd</sup> Dec**

*28<sup>th</sup> Nov: Last day of Michaelmas term*

Implement an interpreter to evaluate the parse tree output by the parser. This interpreter will guide the design of the abstract machine, and may be useful as a performance benchmark.

Milestones:

- Code for interpreter written and tested.
- Document produced to describe the implementation of the interpreter.

### **3<sup>rd</sup> Dec – 16<sup>th</sup> Dec**

Design the byte-code to represent Prolog programs, and write an abstract machine to execute this byte-code. Write the translator to convert the parse tree to byte-code.

Milestones:

- Code for abstract machine written and tested.
- Code for translator written and tested.
- Document produced to describe the byte-code format chosen.
- Document produced to describe how the abstract machine works.

### **17<sup>th</sup> Dec – 23<sup>rd</sup> Dec** [*Slack*]

Read about possible extensions to the project. This could be research on type, mode, or determinacy analysis for Prolog. Of these options I expect type checking to be the most complex, and determinacy analysis the least complex. I will choose where to invest my time depending on how far ahead or behind the project is.

Milestones:

- Produce document to summarise research into optimisations.

### **24<sup>th</sup> Dec – 30<sup>th</sup> Dec** [*Christmas break*]

### **31<sup>st</sup> Dec – 6<sup>th</sup> Jan** [*Slack*]

Continue research into extensions.

Milestones:

- Add more content to the previous summary document.

### **7<sup>th</sup> Jan – 20<sup>th</sup> Jan** [*Slack*]

*17<sup>th</sup> Jan: First day of Lent term*

Implement extension features for the compiler.

Milestones:

- Code for extension features written and tested.
- Document produced to describe the theory and implementation of the extension features.

**21<sup>st</sup> Jan – 27<sup>th</sup> Jan**

Prepare for progress report and presentation.

Milestones:

- Progress report and presentation written.

**28<sup>th</sup> Jan – 10<sup>th</sup> Feb**

**1<sup>st</sup> Feb:** *Progress report deadline*

Write a draft of the structure of the dissertation, giving the main points to be covered. Write a full version of the evaluation section of the dissertation.

Milestones:

- Draft structure of dissertation written, and sent to DoS and supervisor.
- Evaluation section written, and sent to DoS and supervisor.

**11<sup>th</sup> Feb – 24<sup>th</sup> Feb**

Respond to any feedback on the draft structure or evaluation section.

Write the introduction, preparation, implementation, and conclusion sections of the dissertation. I will be able to use material from the documents I intend to produce throughout the project, so I will not have to write these sections from scratch.

Milestones:

- Dissertation completed, and sent to DoS and supervisor.

**25<sup>th</sup> Feb – 10<sup>th</sup> Mar** [*Slack*]

I will use this time to finish the dissertation if the first draft is not already finished. If a draft is finished then I will likely be waiting for feedback, and will dedicate this time to revision.

Milestones:

- If the previous milestone was missed then it should now be met.

**11<sup>th</sup> Mar – 24<sup>th</sup> Mar**

**13<sup>th</sup> Mar:** *Last day of Lent term*

Respond to any feedback on the dissertation.

Milestones:

- Dissertation updated following any feedback received.

**25<sup>th</sup> Mar – 7<sup>th</sup> Apr** [*Slack*]

Continue improving the dissertation and responding to feedback.

Milestones:

- Dissertation updated.

**8<sup>th</sup> Apr – 21<sup>st</sup> Apr** [*Slack*]

Continue improving the dissertation and responding to feedback.

Milestones:

- Dissertation updated.

**22<sup>nd</sup> Apr – 5<sup>th</sup> May**

**25<sup>th</sup> Apr:** *First day of Easter term*

Make any final changes and submit the dissertation.

Milestones:

- Dissertation submitted.

**17<sup>th</sup> May:** *Dissertation deadline*

## Resources Required

I am intending to use my own laptop for the project (Dell XPS 15, 16GB RAM, i7-7700HQ, running Ubuntu). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

The basic resources that are essential to my project are the OCaml compiler `ocamlopt` and the SWI-Prolog interpreter `swipl`. These are both installed on the MCS system. I will be able to complete the project using the MCS machines if my laptop fails.

I will use git for revision control. My git repository will be hosted in GitHub, and this will serve as a cloud backup of my code. I will store my local copy of the git repository inside a Dropbox folder so that my work is also all automatically synced to Dropbox. I will perform weekly backups to a USB stick in case these other backup techniques fail.

## References

- [1] OUnit. <http://ounit.forge.ocamlcore.org/>. OCaml unit testing framework.
- [2] Jan Wielemaker. Prolog Unit Tests. [http://www.swi-prolog.org/pldoc/doc\\_for?object=section\(%27packages/plunit.html%27\)](http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/plunit.html%27)). SWI-Prolog Documentation.
- [3] Memory profiling with Spacetime. <https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html>. OCaml Documentation.
- [4] Execution profiling. <http://www.swi-prolog.org/pldoc/man?section=profile>. SWI-Prolog Documentation.
- [5] Thomas W. Getzinger. The costs and benefits of abstract interpretation-driven prolog optimization. In *Proceedings of the First International Static Analysis Symposium on Static Analysis*, 1994.
- [6] Saumya K. Debray and David S. Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207 – 229, 1988.
- [7] Alan Mycroft and Richard A. O’Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.
- [8] Jason Hickey Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. O’Reilly Media.
- [9] Lexer and parser generators (`ocamllex`, `ocamlyacc`). <https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html>. OCaml Documentation.